Collaborative Identification of Code Smells: A Multi-case Study

Roberto Oliveira¹, Leonardo Sousa¹, Rafael de Mello¹, Natasha Valentim², Adriana Lopes², Tayana Conte², Alessandro Garcia¹, Edson Oliveira², Carlos Lucena¹

¹PUC-Rio, Rio de Janeiro, Brazil, ²UFAM, Manaus, Brazil, {rfelicio, lsousa, rmaiani, afgarcia, lucena}@inf.puc-rio.br, {natashavalentim, adriana, tayana, edson.cesar}@icomp.ufam.edu.br

Abstract—Code smells are considered key indicators of software quality degradation. God Classes and Feature Envy are examples of code smells that frequently become the target of software refactoring. However, smell identification might be harder than expected due to the subjectivity involved in the recognition of the apparently simple structure of each smell. Moreover, smell identification might require the knowledge of multiple program elements, which are better understood by different developers. Thus, the use of collaboration among developers may have the potential to improve effectiveness on smell identification. However, current knowledge, especially empirically developed and evaluated in the industry, is quite scarce. This paper reports an industrial case study aimed at observing how 13 developers individually and collaboratively performed smell identification in five software projects from different software development organizations. Our results suggest that collaboration contributes to improving effectiveness on the identification of a wide range of code smells. We also analyzed the factors contributing to such effectiveness improvement.

Keywords—Case Study; Collaboration; Identification of Code Smell; Qualitative Analysis;

I. INTRODUCTION

As software systems evolve over time, they invariably undergo changes that can lead to software degradation. Code smells are often considered as key indicators of software quality degradation [12]. They affect different code units, such as classes and methods. God Classes and Feature Envy are examples of code smells that frequently become the target of software refactoring [3]. Although there is a considerable set of static analysis tools for code smell detection (e.g. [15][16]), developers still need to validate each smell reported by these tools and confirm its relevance on the system.

Thus, smell identification might be harder than expected due to the subjectivity involved in the recognition of the apparently simple structure of each smell [13][17]. Not rarely, different developers disagree on the incidence of a code smell in a particular code element [13][17]. A comprehensive knowledge of the system plays an important role to mitigate this subjectivity [15][17]. Promoting collaboration among developers may have the potential to enhance smell identification as it allows the smelly structure to be analyzed by two or more developers. In fact, this task might require the knowledge of various program elements, which are better understood by different developers. Therefore, it is important to understand the circumstances that make collaborative identification of code smells worthwhile. However, code smell identification is usually performed through individual reviews in which each professional validates smells reported by a code smell detection tool [14].

This paper reports two executions of an exploratory case study designed to understand the impact of collaboration on the identification of code smells. In each execution, participants were assigned to identify code smells in the systems they implemented. They also received a list of code smells detected by a static analysis tool to be used as a baseline. The study involved five software projects from different domains developed and maintained by two Brazilian software companies, located in the city of Manaus. Thirteen developers of these projects with different experience levels in software development activities had participated. We observed that collaborative identification tended to identify smells more precisely than solo identification. We also analyzed the factors contributing to this improvement. We hope the findings reported in this study shed light on how software development teams can be more effective on the smell identification task.

The remaining sections of this paper are structured as follows. Section [4] presents the background required to understand the work. Section [11] shows the study design. Section [17] presents the results and discusses the main findings. Section [19] presents potential threats to validity. Section [21] concludes the paper with final considerations and an outline of our intended future work.

II. BACKGROUND AND RELATED WORK

Code smells are often considered key indicators of software quality degradation [6]. Examples of types of code smell vary from method-level smells, such as Long Method and Feature Envy, to class-level smells, such as God Class and Shotgun Surgery [5][6]. Code smells can indicate elements that can demand significant maintenance effort [8][24]. Therefore, identifying and removing these smelly elements are important tasks to preserve the software maintainability.
Identification of code smells is not a trivial task. First, this task still has to be manually performed by developers despite the extensive tool support for smell detection available nowadays [15][16]. In addition, after smell candidates are identified by an existing tool, developers still need to analyze each smell individually and confirm its relevance on the system. Second, reasoning about a code smell frequently requires a non-local comprehension of various classes or components somehow related to that smell. Therefore, multiple developers may better understand such program structures when they work together. Thus, it is expected that the collaborative identification of code smells might be more effective than identifications performed individually.

Although the collaboration among developers seems promising, its usage on the identification of code smells is often uncommon, mainly due to two reasons [5]. First, some organizations still believe that this task is more cost-effective when assigned to a single developer [2]. Second, even when software teams are eager to identify code smells collaboratively, they have little information to guide them in this task. Many smells that are possibly better identified through two or more developers may be missed or postponed until when their removal becomes costly or prohibitive in the long run [12].

To the best of our knowledge, we did not find any other investigation on how collaboration can support developers during smell identification tasks. We only found some studies investigating how collaboration can be used to enhance the structure of the source code (e.g., [22][23]), and how the collaboration can be used to increase developers’ productivity [10]. In our previous work [18], we investigated how novice developers can benefit from the collaborative identification of code smells. The findings suggested that collaborative work increases the effectiveness of novice developers when identifying code smells. However, the study consisted of a controlled experiment with novices using software projects in which the professional developers were not involved. Thus, we could not observe developers reviewing the source code of their projects in their actual working environment. The case study reported here addresses these gaps.

III. Study Design

This section presents the design of the case study, also characterizing the context of the organizations involved in both executions.

A. Goals and Metrics

We intend to Analyze the collaboration among software developers; For the purpose of characterize; With respect to the identification of code smells; From the point of view of researchers; In the context of professional developers reviewing the source code of real projects developed by themselves.

We define collaboration when two or more developers work together identifying code smells in the same part of a project. Our definition is similar to the method proposed by Meneely et al. for defect prediction [11]. Based on the presented goal, we established the following research questions (RQs):

RQ1. How collaboration affect the effectiveness of code smell identification?

We observed developers reviewing their projects at their working environment in order to answer the RQ1. We asked them to identify code smells in two scenarios. In one scenario, developers had to work individually on the identification task. We say that a developer work individually on the identification of code smells, when he has to work alone in the task. In the other scenario, developers had to work collaboratively on the identification task. We say that two or more developers work collaboratively on the identification of smells when they work together in the task.

We used precision and recall metrics to compare developers’ effectiveness in both scenarios (individually and collaboratively). We established a reference list of code smells to calculate these metrics. The reference list is composed of the code smells identified by the project manager of the system and two code smell researchers, as explained in Section III-F.

Precision and recall were calculated based on the number of code smells marked as true positive (TP), false positive (FP) and false negative (FN). A TP occurs when the developer identifies a code smell that is also included in the smell reference list. A FP happens when the developer identifies a code smell that does not match any smell in the reference list. A FN occurs when a code smell is present in the smell reference list, but the developer did not identify it. We rely on formulas commonly applied to compute precision and recall, available at [19]. High recall values (close to 1) indicate that the developer was able to identify a representative set of TPs in the system. High precision values (close to 1) means that the developer had reported, proportionally, few FPs.

We also used the precision and recall to compare the developers’ effectiveness in a third scenario. In this other scenario, the developers worked individually on the identification of code smells as aforementioned. After that, we aggregated its precision and recall with the precision and recall of a second developer. This second developer corresponds to the developer whom he worked collaboratively. Thus, for the developers that we aggregated their results, we say that they worked cooperatively on the identification of code smells.

After answering our first research question, we observed in depth how developers identify code smells. That leads us to the second research question:
Table I
OVERVIEW OF THE COMPANIES COVERED BY THIS CASE STUDY

<table>
<thead>
<tr>
<th>Features</th>
<th>Co1</th>
<th>Co2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of company</td>
<td>Public</td>
<td>Private</td>
</tr>
<tr>
<td>Number of employees in project</td>
<td>80</td>
<td>150</td>
</tr>
<tr>
<td>Type of software</td>
<td>Information systems</td>
<td>Information systems</td>
</tr>
<tr>
<td>Type of domains</td>
<td>Government administration</td>
<td>Industrial automation</td>
</tr>
<tr>
<td>Development process</td>
<td>Prescriptive</td>
<td>Prescriptive and Agile</td>
</tr>
<tr>
<td>Programming language</td>
<td>Java</td>
<td>Java, Android, iOS</td>
</tr>
<tr>
<td>Agile development methods</td>
<td>-</td>
<td>Kanban and Scrum</td>
</tr>
<tr>
<td>Platform for inspection of code</td>
<td>-</td>
<td>SonarQube</td>
</tr>
<tr>
<td>Code Review</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

RQ2: How do professionals identify code smells?

This question was created aiming to understand how professional developers perform the identification of code smells in the industry. We want to observe developers performing this task in two scenarios: working individually and working collaboratively. Based on this observation, we expect to identify the main benefits and drawbacks of performing code smell identification in both scenarios.

B. Software Companies

After discussing alternatives of companies to be involved in the study, we opted by selecting two Brazilian software companies. We selected them based on their different features, including the level of experience with code reviews, the number of developers, application domains, and development process. Table II summarizes these features. We opted by running the case study with two companies, from now on called as Co1 and Co2, to balance contextual diversity with convenience [20]. Thus, we conducted a specific instance of the case study in each company: CS1, performed in Co1 and CS2, performed in Co2.

These two companies showed interest in identifying code structures that reduce the software maintainability like the code smells. That is the other reason that motivated us to selected them. In the first meeting with these companies, it was clear that they cared about the system quality. Moreover, they were excited to spread to the developers the knowledge on a relevant topic for the system maintainability. In fact, one of these companies has the culture of holding seminars and training. They maintain this culture to propagate knowledge on topics and technologies that are relevant to some developers in the company but are unknown to others.

Together with the companies, we searched for systems that (i) have been developed in Java, (ii) with different sizes (lines of code), and (iii) from different domains. Based on this search, we selected five systems for this study. Two systems from the company Co1 and three systems from the company Co2. The selected systems have a different number of developers in each software maintenance team, varying from three to seven developers. Thus, we asked the project manager of each system to indicate the developers to participate in the study.

1) CS1: a government company

The first execution of the case study (CS1) was performed in Co1, a government organization that develops software to manage and control the Brazilian Government budget. Such company recently started to apply code review sessions in their projects. After a meeting with the company representatives, we selected two critical systems developed and maintained in Co1 for more than seven years. The first system (S1) controls the entrance of any product in Amazonas state (a Brazilian state) and processes tax revenues. The second system (S2) standardizes budget reports in the same state. As those systems handle with the government budget, problems in these systems may lead to severe impact in the government accountability and budget. Table II summarizes the background of the seven participants (p1 to p7) selected in CS1, distributed by the system currently maintained by them. Most of the participants are bachelors in Computer Science, having at least four years of professional experience in Java programming. Only one participant reported some previous experience on performing collaborative code reviews.

2) CS2: a private company

The second execution of the case study was performed in Co2. This company is a private non-profit foundation that has national and international customers. Co2 is a technological center for software and hardware research and development. Although Co2 project teams are free to define how they manage the system quality, some teams have managed the system quality through code reviews. After a meeting with the company representatives, we selected three systems developed and maintained by the company. The first system from Co2 (S3) supports the management of registry offices for audition and control in the Amazonas State’s Court of Justice. The second system (S4) is a computational solution for maintaining patients’ historical information in...
a hospital. It uses electronic medical records to integrate patients’ clinical and administrative information. The third system (S5) is a system developed to trace products in their production lines. This system allows clients to track products from its origin to retail locations.

Table III summarizes the background of the six participants (p8 to p13) selected in CS2, distributed by the system currently maintained by them. Most of the participants are bachelors in Computer Science. All of them have three years or more of professional experience in Java programming. On the other hand, most of them had some previous experience in performing code review collaboratively.

C. Instruments

We used four sources to collect data: participants characterization questionnaire, code smell report, task execution records and a follow-up questionnaire. The participants’ characterization questionnaire is composed of questions to characterize each participant, including academic degree, professional experience with Java programming, and background on code review activities (with or without collaboration). The code smell report is an online form used by the participants to list the code smells identified during their tasks. The task execution records includes the screen shots captured by Camtasia as well as audio and video streams capturing all participants activities. Finally, we applied a follow-up questionnaire to collect the participants’ impressions regarding the code smell identification tasks. The follow-up questionnaire is available at [19].

D. Procedures

The study was composed by a set of four activities distributed into three phases, as represented in Figure 1 described as follows.

Activity 1: Apply the questionnaire for participants’ characterization. As previously mentioned in Section III-C, a questionnaire was designed to gather developers’ information on several aspects.

Activity 2: Training. This activity aimed to even up the knowledge of participants through a tutorial. The tutorial addressed definitions and examples of code smells as well as the relevance associated with their identification. The first and second authors applied the same training session for participants in each company. The training was organized in two parts. The first part devoted to explaining the technical concepts involved in the study (25 minutes). The second part was dedicated to discussion (10 minutes). However, we extended the second part whenever the participants had additional questions or concerns to be addressed.

Activity 3: Smell identification task. We asked the participants to work through two rounds of code smell identification. They worked on the same system in both rounds, but in disjoint sets of modules in each round. Figure 1 illustrates the identification task. In the first round, the developers had to identify the code smells individually in a set of modules (e.g., S1 in the Figure 1). In the second round, they had to identify the code smells collaboratively, but in other set of modules. (e.g., S1′ in the Figure 1). Table III presents the arrangements of participants according to their systems. Participants received a list of code smells from 15 different types detected by an automated tool to be used as a baseline (Section III-F). In this sense, participants were oriented to feel free on following or not the given list. Participants also received a guide that characterizes each type of code smell used in this study. Each round lasted 45 minutes.

Activity 4: Answer the follow-up questionnaire. The participants were asked to individually answer a questionnaire about their experience on code smell identification. A follow-up questionnaire was distributed to the participants at
the beginning of Activity 3, but they fully answered it after they concluded their study tasks. This strategy was intended at helping participants on providing detailed feedback.

E. Data Analysis

We performed the quantitative data analysis as explained in Section III-A. We compared the effectiveness obtained by developers identifying code smells individually and collaboratively. To better understand the impact of the number of developers in the results, we also compared the collaborative results of each project with the aggregated results of the individual reviews performed in the same project. We called such aggregation by cooperative review.

The qualitative data analysis over the records of video and audio was based on the procedures of Grounded Theory (GT) suggested by Strauss and Corbin [21]. We used open coding (1st phase) and axial coding (2nd phase) from the GT method. Open coding involves the breakdown, analysis, comparison, conceptualization, and the categorization of the data. Axial coding examines the relations between the identified categories. When analyzing the qualitative data, we created codes for the developers speeches (1st phase). After, these codes were related to each other - through axial coding (2nd phase). Finally, selective coding performs all the process refinements by identifying the core category to which all others are related. We decided not to select a core category herein because a Grounded Theory rule is which all others are related. We decided not to select a core category herein because a Grounded Theory rule is the circularity between the collection and analysis stages until the theoretical saturation is reached [21]. Therefore, we decided to postpone the selective coding phase. For this reason, we do not claim that we applied the GT method, only some specific procedures.

The first and fourth authors did the open coding, associating codes with quotations of transcripts, and axial coding, at which the codes were merged and grouped into more abstract categories. For each transcript, the codes and identified networks (memos showing the relationships in the categories) were reviewed, analyzed and changed upon agreement with the others researchers.

The phases of the open and axial coding were sufficient to identify the strategies that developers use to identify code smells as well as the opportunities to improve this task. These procedures allowed us to emerge different strategies followed by developers to identify code smells. We also performed free content analysis over the answers given by the participants to the follow-up questionnaire.

In order to get this data, we run the two study instances sequentially. Each instance was executed in the respective developers’ working environment. In total, the 13 participants worked 14 hours and 25 minutes in the identification tasks, reporting 222 code smells. From this time, four hours and twenty-five minutes were devoted to collaborative identification. We generated 593 codes after performing the 1st phase of the GT method and established 584 associations.

F. Reference List of Code Smells

We selected 15 types of code smells to be identified in the study. These types were selected due to several reasons. First, these code smells are both: (i) very common in the chosen projects, and (ii) typically associated with software degradation symptoms [7][12]. Second, some of these smells are often subsumed with other ones (e.g. God Class can be subsumed with Long Method and Feature Envy). Consequently, they may impose different levels of difficulty to the identification task. Finally, these smells demand the analysis of various structural characteristics of the program elements, such as size, coupling, cohesion, complexity, among others [6]. Details about the code smell types and their descriptions are available in our complementary material [19].

We contacted the project manager of each system to create the reference list of code smells. First, we run a detection tool to produce an initial list of code smells of each project used in the study. The tool is based on a set of previously defined rules for smell detection [11], available in [19]. These rules were successfully applied in different studies [9], achieving high precision. After creating the initial list, we manually validated each code smell candidate suggested by the tool. The manual validation was performed in two steps by two researchers (first and second authors) and the system project manager. In the first step, the two researchers classified each code smell detected by the tool as TP or FP. In the second step, they conducted an open discussion with the project manager to solve any conflict and to reach a consensus. We highlight that some code smells were also added after the study. That happened when we analyzed the developers’ answer, and we verified that some of the smells reported by them, but not found by us, should be included in the reference list.

IV. RESULTS

This section presents the quantitative and qualitative results to answer both research questions. Section IV-A presents the quantitative results of each company. Sections IV-B and IV-C combine the quantitative and qualitative analysis to answer RQ1 and RQ2, respectively.

A. Quantitative Analysis

This subsection presents the quantitative data of each study execution. We present the results of each round, i.e., when the developers worked individually, and when they worked collaboratively on the identification of code smells.

1) CS1: Developers from the first organization

Table IV shows the results of the first round in CS1. The table shows the TP, FP and FN measures associated with each developer when working alone. Each developer, represented by his ID (2nd column), worked on a specific system (1st column). The last columns present precision and recall results. Table V follows a similar structure. However, it presents the results of the second round in the same
company. Thus, the 2nd column shows which developers worked collaboratively. The four developers, who worked on the S2 system in round 1, were divided in two pairs in round 2. The three developers of the S1 system worked together as a group in round 2.

Comparing the precision and recall of Tables IV and V, we observe the following trend: precision was consistently improved (except for p2) in round 2, while recall was slightly improved. We did not expect any improvement on recall given the time constraints. The three developers, who worked as a group, did not achieve clearly better results than their individual performances, except for recall.

As mentioned in Section III-A, we also aggregated the results of the developers who worked individually on the same project during the identification of code smells in round 1. In particular, each aggregation in round 1 was computed for those who would work together as a pair or a group in round 2. In other words, each aggregation is fruit of their cooperation in round 1, i.e., it represents the joint result of the individuals’ performance. Table VI presents the results of such cooperative performance in round 1. These results allow us to observe whether the collaborative identification of smells in round 2 outperformed (or not) the cooperative identification in round 1.

The comparison of results in Tables IV and VI reveals that developers’ collaboration consistently achieved higher precision than their cooperation. On the other hand, cooperation achieved a higher recall than collaboration, except in one case. Surprisingly, when we analyzed the code smells identified by the developers working cooperatively, no one found the same instance of code smell in a particular class. In other words, there was no intersection of code smells reported by two or more developers working individually. This phenomenon will be discussed in Section IV-C.

2) CS2: Developers from the second organization

Different from CS1, developers from CS2 worked collaboratively in round 1 and individually in round 2. Tables VII and VIII present the results achieved by each group and each developer, respectively. According to these results, precision and recall were consistently better (except for p12) in their collaborative performance as compared to their individual performance. P12 achieved similar results in both rounds.

Table IX presents the aggregated results of the cooperative performance of developers in round 2. As occurred in CS1, no intersection in the sets of code smells reported by participants was found. Comparing the results in Tables VII and IX, one can observe developers reached much higher precision working collaboratively than cooperatively. On the other hand, two groups obtained slightly higher recall when working in cooperation.

B. Effects of Collaboration on Smell Identification

We compare the individual and collaborative performances of developers in order to answer RQ1: How collaboration may affect smell identification effectiveness? As explained in Section III-A, we used precision and recall metrics to compare developers effectiveness in both situations. After analyzing the results from both companies (Tables IV, V, VII and VIII), we noticed some similar trends in both case study executions.

First, almost all developers from both companies reached better precision and recall when they worked collaboratively than individually. Only one developer (p2) from the Co1 achieve better precision when he worked individually. We did not observe any effect of swapping the order of the individual and collaborative tasks along the two executions. In other words, collaborative identification outperformed individual identification of smells in both executions.

Second, we observed Co2 developers reached better precision and recall results working collaboratively than Co1 developers. Analyzing the characterization forms of all developers (Section III-B), we noticed that Co2 developers had previous experience with peer review, while Co1 developers had none. This experience likely helped them to better explore the benefits of collaborative identification of code smells. These results lead us to our first finding:
Table VIII
RESULTS: developers working individually in the CS2

<table>
<thead>
<tr>
<th>System</th>
<th>ID</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3</td>
<td>p8</td>
<td>5</td>
<td>11</td>
<td>30</td>
<td>0.31</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>p9</td>
<td>5</td>
<td>14</td>
<td>30</td>
<td>0.26</td>
<td>0.14</td>
</tr>
<tr>
<td>S4</td>
<td>p10</td>
<td>13</td>
<td>5</td>
<td>8</td>
<td>0.72</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td>p11</td>
<td>5</td>
<td>8</td>
<td>16</td>
<td>0.38</td>
<td>0.24</td>
</tr>
<tr>
<td>S5</td>
<td>p12</td>
<td>3</td>
<td>8</td>
<td>27</td>
<td>0.50</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td>p13</td>
<td>2</td>
<td>7</td>
<td>28</td>
<td>0.22</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table IX
RESULTS: developers working cooperatively in the CS2

<table>
<thead>
<tr>
<th>System</th>
<th>ID</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3</td>
<td>p8 + p9</td>
<td>10</td>
<td>25</td>
<td>25</td>
<td>0.29</td>
<td>0.29</td>
</tr>
<tr>
<td>S4</td>
<td>p10 + p11</td>
<td>18</td>
<td>13</td>
<td>3</td>
<td>0.58</td>
<td>0.86</td>
</tr>
<tr>
<td>S5</td>
<td>p12 + p13</td>
<td>5</td>
<td>15</td>
<td>25</td>
<td>0.25</td>
<td>0.20</td>
</tr>
</tbody>
</table>

Finding 1. Developers working in collaboration tended to be more effective on smell identification than solo developers.

As far as recall is concerned, we noticed that the results were not high in general. This behavior was somehow expected given usual time constraints of code review sessions. We noticed that developers clearly tended to focus only on smells they consider as critical. Moreover, one could argue that developers’ cooperation (Tables VIII and IX) slightly outperformed developers’ collaboration (Tables VII and VII) in terms of recall. In principle, results might suggest collaborative smell identification tend to be more precise but less comprehensive (recall) than cooperative identification.

However, we observed that individuals produced fewer FN only because they tended to identify much simpler smells (Section IV-C) when working in isolation. In contrast, teams were able to identify more complex smells (Section IV-C), which remained unnoticed by individuals. Moreover, developers avoided to make mistakes (false positives) when working with somebody else. This was confirmed by the analysis of the qualitative data. All the aforementioned results lead us to the first recommendation:

**Recommendation:** Companies should encourage collaborative identification of smells in order to reveal critical refactoring opportunities and avoid unnecessary ones.

C. Identifying Code Smells

Analyzing all the discussions among developers and, based on the quantitative analysis and the answers obtained from the follow-up questionnaire, we could answer our second research question: *How do professionals identify code smells?* We applied the qualitative analysis to identify the actions performed by the developers in both rounds. Moreover, we checked if those developers’ actions contributed to the accurate identification of code smells (true positives) or otherwise (false positives).

**Smells and Comfort Zone.** As presented in Section IV-A when we analyzed the code smells identified by the developers, no one found the same code smell in the same class. There was no intersection of code smells reported by two or more developers. It can be explained by the observed trend of developers focusing their analysis on the code which they had worked in the past. In fact, we noticed that developers working individually tend to stay in their comfort zone concerning the analyzed classes. In most of the cases, they analyzed only the classes that they knew. As a result, they often identified simpler smells internal to the class under their ownership, such as Long Methods.

**Leveraging Complementary Knowledge.** On the other hand, when they worked collaboratively, they were keen to analyze other classes that his teammate indicated. As soon as they shared their knowledge about different classes, they started to reveal more complex smells affecting multiple classes. For example, p1, p2 and p3 were analyzing ClassA during the search of a Long Parameter List smell in CS1. When they were analyzing ClassA, p1 reported that a piece of code in that class was duplicated from the ClassB. As p1 noticed the duplication, they confirmed these classes were embodying the Duplicated Code smell. Thus, p1 shared his knowledge about the ClassB with the other two developers.

This situation is reported below:

“Guys, let us go first to ClassB. Methods in the ClassB are duplicated” – p1
“Now, go to the ClassB.setZ method... its code is duplicated. This piece of code is the same from ClassA.setY.” – p1

**Recommendation:** Companies should encourage collaboration and include another developer to help developers to identify code smells in the other classes.

**Recommendation:** Companies should encourage collaboration and include another developer to help developers to identify code smells in the other classes.

This example illustrates a scenario in which collaboration helped the developers to identify a smell in code elements that they did not have knowledge. A developer may not know about the entire system, but when he teams up with another developer, they can benefit from the individual knowledge of each other. Thus, they can identify code smells that require a broad understanding of the system like exemplified. During the analysis, we found more cases of code smells that were identified only when the developers teamed up. Most of those cases are related to Duplicated Code, God Class, Shotgun Surgery, Refused Bequest and Speculative Generality. These types of code smells often require a
Identification of God Classes required developers to check if each suspicious class fulfills more than one responsibility. For example, developers p1, p2, and p3 were investigating the InvoiceInput class. They were discussing whether the class was a God Class or not. They mentioned that in their systems, the report generation is related to several classes - no class is responsible for generating all the reports. Otherwise, such class would be a God Class. Then, they realized that the ReportGenerationAction class was generating different types of reports. Thus, the developers indicated that the ReportGenerationAction class was a God Class.

“But if we select this (InputInvoice) class, you have to start with God Class since this class has all the types of reports. Those total (attributes) are all from here.” – p2

“Yeah, God Class and Feature Envy smells.” – p3

“Yeah, but I wonder if this class is a God Class, what if the report (generation) is related to the InvoiceInput class?” – p1

“Yeah, the issue we have here is that we won’t have a report (generation) that is related to only one class. It (the report generation) always includes much others (classes). For example, the tax, the invoice, the item. What should we do?” – p2

“Do we need to create a separate class for that?” – p1

“If we create a class for report (generation)... but I’m not sure.” – p2

“By the way, there is a class in charge of generating reports.” – p3


“No, but it (the class) is a action, it is not a DAO.” – p2

“And it is not being considered as a class.” – p1

“Yeah, but look, this class is a God Class. Besides God Class and Feature Envy, what else do we have?” – p3

God Class and Duplicated Code seem to be code smells that require a broader knowledge of the software system. Sometimes, developers need to reason about multiple elements before confirming if the element has a code smell or not. Thus, given the need for global reasoning, information exchange among two or more developers may help them to identify better particular types of code smells. Therefore, the number of developers involved in smell identification may affect task effectiveness. This result leads us to the second recommendation:

**Finding 2.** Developers working collaboratively benefited of shared knowledge to identify code smells that require a broad understanding of the system.

**Recommendation:** Companies should promote developers’ collaboration for improving identification of smells requiring non-local reasoning.

**Contextual Information.** Finally, we noticed that teams needed contextual information to make well-informed decisions about each smell occurrence. The God Class case shows that the identification of code smells is not limited to analyze a set of metrics and thresholds related to the suspicious class. Developers had to verify if the class fulfilled more than one responsibility. The understanding of each responsibility may require the analysis of various classes realizing this responsibility. In other words, developers needed to analyze a wide range of contextual information to properly identify code smells, leading to our third finding:

**Finding 3.** Developers needed a wide range of contextual information before making a decision about the code smell.

After we had analyzed the qualitative data, we classified the contextual information frequently mentioned by developers in four categories, which are presented below.

1) **Surrounding Information:** Sometimes the developers need to analyze the elements around the class that contains a code smell. For example, to confirm a Feature Envy, it is also necessary to inform which methods and attributes that the envy method accesses in other class. Also, developers needed to check whether the class, which the method seems to be interested, should receive the envy method or not. Moving the method would also have implications to the clients of the target class.

2) **Historical Information:** The developers used the historical information of an element to understand its evolution. They often tried to understand what happened with the element since its creation. For example, they were trying to figure out what happened with a class that used to be affected by a method-level code smell but now it is affected by a class-level smell. Developers also tried to reveal through the history when the class became a God Class, i.e., when it started to implement other functionalities:

“It became a complex class over the time since it was used for several other things, including various other types of verifications. Formerly, it was only a class used to verification, but then it became integrated into many other classes as quota request and debt recognition. Today, it is also used to load a part of the pledge. Thus, it began to serve...
3) **Developer Information:** Developers may use the information about other developers that contributed to the implementation of each class. Thus, they can exchange information about the classes in order to confirm or refute the existence of a code smell. In the following, we present a situation in which developer’s knowledge about the class was essential to avoid a false positive involving a Duplicated Code smell. In this case, the developers p2 and p3 do not know much about the analyzed class. However, p1 helped to implement the class. Thus, he uses his knowledge of the method to explain why it should not be considered as a Duplicated Code.

“This method is so long. I can’t understand what it is doing.” – p2

“Here’s the thing… this method has two implementations. The first one uses a rule, but the rule changed after a certain data. So, all the entities created before the data use the first implementation of the method, and all the entities created after the data use the new implementation of the method. We have this problem; the code has a temporal rule. This is crazy.” – p1

“So it’s not considered a Duplicated Code” – p3

“No. It’s not a Duplicated Code” – p1

4) **Framework Information:** Developers also need information about the framework used in the project. This information may help the developers to understand why the element has a smell or not. If the developer knows about the used framework, he can configure the code smell detector. Thus, he can avoid false positives generated because the developer used a framework. For example, during the identification task, p8 and p9 developers noticed the importance of being aware about the framework. In this case, p8 mentioned that identify code smells automatically is not an easy task because there are some code smells that require to understand the context of the element. He mentioned that the element should not be considered as having a code smell because the framework forced the developer to implement in that way.

“It (the class) is doing what it supposed to do. It is sending what it supposed to send.” – p8

“But the reason (of the code smell) here is the framework (...)” – p9

“This element is hard to detect correctly because of the framework. We have to understand the context here. The reviewer needs to understand the context.” – p8

These examples of contextual information mentioned by the developers lead us to the third recommendation:

**Recommendation:** Strategies to support collaborative identification of code smells should provide contextual information.

V. **Threats to Validity**

**Construct validity:** We highlight two threats to construct validity concerning to the case study plan: (i) the distribution of code smells in each project, and (ii) the composition of the reference list. In order to mitigate (i), we selected classes of each project composed by similar distributions of these code smells. In order to mitigate (ii), the composition of the reference list had involved both researchers and software managers from each project. The set of 15 code smells used in the study covered smells at different levels of granularity. Moreover, there is empirical evidence that such code smells are associated with varying degrees of maintenance effort (Section III-F).

**Internal Validity:** The time restriction to conduct the identification of code smells can be considered a threat. We estimated that the developers would have a chance to finish the identification task. This estimation was the result of a pilot phase that we run before this study. Based on the experience of the pilot phase and on the time constraints of the organizations, we calculated that the time limit of 45 minutes would be sufficient to participants identify a considerable list of code smells. It is noteworthy that the pilot also allowed us to identify opportunities to reduce possible discouragement of the developers during the identification tasks.

**External Validity:** Although it is expected from case studies to observe in the small, the execution of different instances of the same case studies can be useful to strength the findings. The limited diversity of contexts involved in the case studies can be considered a threat to validity. However, we argue that the selected companies represent typical software development organizations in Brazil and elsewhere. We described their profile in detail, thus others can understand the contexts related to the both study: CS1 and CS2.

**Conclusion Validity:** This threat concerns the relation between the treatment and the outcome. To mitigate threats to conclusion validity, we planned different methods and instruments for collecting both quantitative and qualitative data. Thus, it was possible to triangulate evidence emerged from the practice, researchers observations and participants opinion, strengthening the study findings.

VI. **Conclusion**

In this paper, we investigated the impact of the collaboration on the identification of code smells. We run a multi-case study involving five software projects and 13 professional developers from two software development organizations. We used quantitative and qualitative analysis during the investigation to answer two complementary research questions.
The results suggest that developers working collaboratively on the identification of code smells tend to be more effective than developers working individually. In fact, when developers worked collaboratively, they benefited from the shared knowledge to identify code smells that require a broad understanding of the system. These results suggest that companies should encourage collaboration between developers to increase the rate of success on the identification of code smells. Moreover, the results also indicate that the collaboration helped developers to reduce the number of mistakenly identified code smells during the identification task.

Analyzing the qualitative data, we also noticed some contextual information that the developers need before making a decision about the code smell. Examples of contextual information include the surrounding context of the affected element, the historical information of a element and the developer information. Future steps of our research involve the planning and execution of new studies to investigate means for better assisting collaborative identification of code smells.

REFERENCES


